Ferran Manuel Boán-Montenegro Albelda

Definition and measurement of KPIs in SaaS

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.
Espoo 21.5.2013

Thesis supervisor:

Prof. Heikki Saikkonen

Thesis advisor:

M.Sc. (Tech.) Lauri Heiskanen

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Ferran Manuel Boán-Montenegro Albelda

Title: Definition and measurement of KPIs in SaaS

Date: 21.5.2013 Language: English Number of pages:7+43

Department of Computer Science and Technology

Professorship: Software systems Code: T-106

Supervisor: Prof. Heikki Saikkonen

Advisor: M.Sc. (Tech.) Lauri Heiskanen

Software as a service (SaaS) is a software delivery model that has consolidated in the past years. With cloud computing and the help of the Internet, this business model is gaining ground as an alternative to traditional software models.
Software quality models such as the ISO/IEC 9126, provide models and metrics to guarantee the correct operation of software.

This thesis aims to provide a set of indicators, based on three aspects: The software quality models and the metrics described in the ISO/IEC 9126, and the characteristics that must exist in SaaS applications.

As a practical exercise we will show how these indicators can be utilized in an existing SaaS application.

Keywords: SaaS, Quality, Web applications, Software as a Service, Treasury, Key performance indicators, KPI

Preface

This is the time to formally thank all the people that has supported me during the years of my studies.

First and foremost I would like to thank my family for their unconditional support despite living in different countries and to my partner, Jeanette, for her patience during the writing process of this thesis.

I would also like to thank my employer, Exidio, for the flexibility that has allowed me to work and study during all these years.

Thanks to my instructor, Lauri Heiskanen for begin around when needed and for his help when reviewing this thesis.

Lauttasaari, Helsinki, May 21, 2013

Ferran Manuel Boán-Montenegro Albelda

Contents

List of Tables

List of Figures

1 Introduction

Software models are constantly evolving. Since the year 2000, the amount of users has quintupled to reach in the year 2010 an estimated amount of 1,9 millions of users [4]. Users are no longer required to have software installed in their computers, instead a web browser and an Internet connection is all that is needed to utilize software. This is one of the reasons why applications based on the SaaS model are consolidating as strong alternatives to traditional software models. On an article by Michael A. Cusumano [6], he highlights the benefits of utilizing SaaS platforms as alternatives and, although they do not seem to be replacing traditional software any time soon, they will overcome this differences sooner than later.

An important part of traditional software applications is quality [14], and in SaaS applications this is no different. However, and as it will be explained later in this work, SaaS applications add extra characteristics that are not present in traditional software or do not carry the same importance. Multi-tenancy and availability are an example of characteristics that are highly relevant in SaaS systems. This creates the need for improving the quality model used in traditional software systems, in order to accommodate these added characteristics.

With the growth of SaaS applications and the need to guarantee a good level of performance in them, there is a necessity to introduce indicators. This thesis introduces a set of KPIs keeping the characteristics of SaaS applications as a goal and using the existing standard for quality defined in the ISO/IEC 9126.

As the practical exercise, the proposed KPIs presented in this thesis will be utilized on an existing SaaS application, Trezone [7].

1.1 Thesis structure

This thesis is divided into six chapters with the first one being this introductory chapter.

The second chapter contains the first theoretical study of this thesis and will introduce the concept of software as a service. The concept will then be used to explain the attributes that drive the design of applications on this model. A maturity model will be presented to explain the possible levels a SaaS system can contain. On the final part of this chapter the use case of this thesis, Trezone, will be explained.

Chapter three contains the second theoretical study of this thesis. It includes the study of software quality and the metrics to measure it. The characteristics presented in the second chapter are combined with the metrics and the software quality models defined in the ISO/IEC 9126 to set the bases of the KPIs introduced in chapter four.

In chapter four the goal of this thesis is presented. A set of key performance indicators are defined based on the research information gathered in chapters 2 and 3.

In chapter five, the KPIs defined in chapter 4 are utilized in combination with the SaaS application defined in chapter 2, Trezone.

Chapter six summarizes the content of this thesis, explains what was learned in this work and discusses future ways this work could be extended.

2 Software as a Service

2.1 Definition

Software as a Service (SaaS from now on) can be defined as delivering software applications over a network. We find a more formal way of defining it [9]:

> "Software deployed as a hosted service and accessed over the Internet"

According to [9], software as a service can be categorized as a line-of-business services and customer-oriented services.

Line-of-business services are paid on a subscription basis. This include SaaS applications such as Spotify or Dropbox. This will be the model that will be further developed in this work.

Customer-oriented services are, on the other hand, services that are paid by advertising. This means that they have no cost for the end user, but are supported by advertising. Google Mail (GMAIL) and Facebook are two examples that fall into this category.

Another formal way to define what software as a service is, is extracted from [18]:

> "Software-as-a-Service (SaaS) applications are based on a recurring subscription fee and typically are a pay as you go model.[...]A typical SaaS deployment does not require any hardware and can run over the existing Internet access infrastructure.[...]The SaaS vendor assumes all the support, training and security risks in exchange for the recurring subscription fees."

There are three key elements that can be extracted from this definition:

- Recurring subscription fee: Fees are paid on a pay as you go model. Among other things, this means that customers who are not satisfied with the service can, at any point in time, cancel the subscription.
- No need for additional infrastructure: This allows SaaS projects to be affordable as it does not require any upfront cost.
- Delegation of responsibilities: By shifting the responsibility of maintaining and supporting the SaaS, the amount of IT resources that a customer needs to allocate for a SaaS system are reduced to a minimum. Allowing them to concentrate on other critical business parts.

2.2 SaaS Architecture

According to [9] there are three attributes that need to exist to consider a SaaS application properly designed: Scalability, multi-tenancy and configuration.

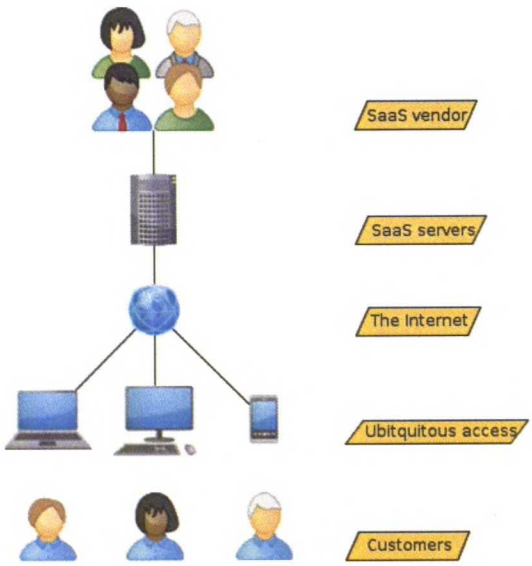

Figure 1: An overview of a typical SaaS system.

2.2.1 Scalability

Scalability of a SaaS system refers to the efficient utilization of resources, as well as the ability to escalate the system to accommodate user needs. Kevin Holmes from rackspace [11], proposes two types of scalability: Horizontal and Vertical.

- Horizontal scalability allows a SaaS application to add more servers similar to what it already exists. This would be equivalent to replicating an existing machine and adding it to the server farm. Figure 2 depicts a picture of this type of scalability.

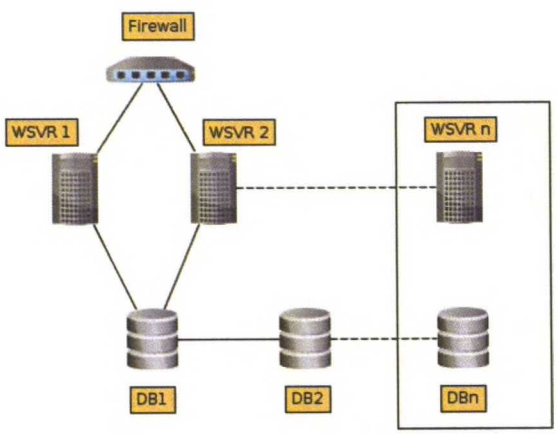

Figure 2: Diagram showing horizontal scalability.

- Vertical scalability allows a SaaS application to scale the servers that are not performing as expected. In practice this would mean to increase disk space, introduce a faster processor or add more memory. Figure 3 depicts a picture of this type of scalability.

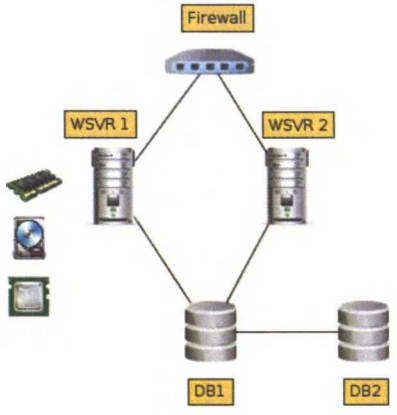

Figure 3: Diagram showing vertical scalability.

2.2.2 Multi-tenancy

In a multi-tenancy environment [10], isolation is a critical to guarantee the privacy of its users. Because of the nature of the architecture, users will share the same application and will use the same servers. In this scenario, guaranteeing customer privacy becomes a challenge that needs to be faced keeping in mind the following considerations:

- Economic considerations: The cost of developing an application that will be shared among many customers is higher than an application that will only be used by only one customer. This is due to a longer process in developing this type of application. Looking at the figure in the long run, developing a shared application is more cost effective than developing an isolated approach.
- Tenant considerations: According to [10], four different questions can be asked to determine what approach is more convenient.
  - What is the **amount of prospective customers** that the application is expected to serve? In cases where the number of tenants is large, it is recommended to choose a multi-tenant approach instead of an isolated.
  - How **much data is each tenant** going to utilize? If tenants are expected to create large databases, it is advised to strive for isolated databases environments.

  - How **many concurrent end users** are going to utilize each tenant instance? The more concurrent users, the more isolated the environment will have to be. Typically in a shared environment where many concurrent users make use of the application, results in the application performing poorly.
  - **Tenant-specific services** such as the possibility to restore a database at any given point in time benefit tremendously from isolated databases environments.

- Regulatory considerations: Sometimes large organizations might require certain security and storage needs. The decision will depend on the type of market the application is going to target. If the application is going to hold sensible data such as personal data or financial data(such as the SaaS application presented in this work, Trezone), it is important to keep databases isolated to guarantee the highest level of privacy.

- Skill set considerations: They are tightly related to the team designing the architecture of the application. Implementing a multi tenant environment is more complex than an isolated and there is more room for mistakes that might compromise privacy in the application.

2.2.3 Configuration

SaaS architecture limits the level of customization an application can have. The primary reason is that all the customers share the same environment and it is an added difficulty to customize it for one particular customer. A potential solution is to introduce metadata that will be customer-specific. Examples of these customization are user interfaces. They can be easily customized by utilizing custom made CSS templates.

2.3 The Maturity Model

Chong [9] introduces a maturity model for SaaS applications. This model aims to provide SaaS architects with flexibility to introduce the previously mentioned attributes scalability, multi-tenancy and configuration to suit their needs.

The underlying idea is that when developing a SaaS application, these three attributes can be used to satisfy architectural needs. The proposed maturity model is divided into four separated levels:

- Level I: Ad Hoc/Custom
  The first level of maturity is considered to be the most simplistic. Essentially it is not different from a self-hosted client-server application. It does not introduce any SaaS benefits, but it allows the software vendor to centrally manage and administer the system.

- Level II: Configurable
  At the second level of maturity, the vendor introduces a change and unifies the

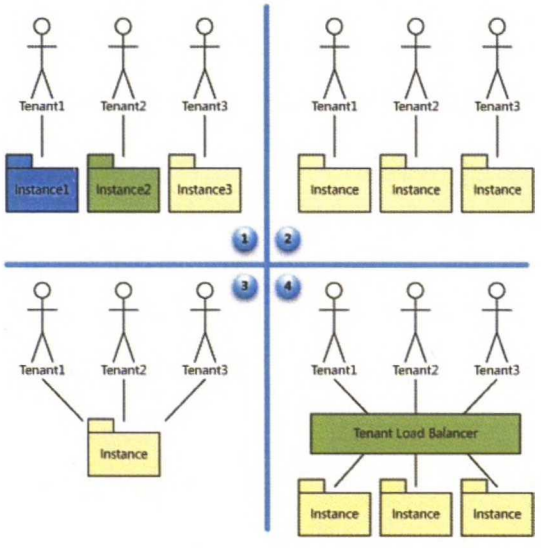

Figure 4: A proposed maturity model for SaaS [9].

code in the instances. This means that the users will effectively be using the same code and the same application, but will not share them. This makes it easier to maintain the code because the same code is utilized by the customers, and upgrades can be executed without having to consider different, customized versions. It is important to note that at these two first levels, the vendor has to allocate resources to support the execution of multiple instances.

- Level III: Configurable, Multi-Tenant-Efficient
  At the third level of maturity, the introduction of multi-tenant-efficient term brings optimization of resources. This is due to the fact that users share the same hardware resources. On the application level, the visibility still stays the same which translates into customers not being aware of the fact that physical resources are being shared. From the customer point of view it gives the impression that each customer is running an independent instance, but from the vendor point of view it allows to unify the hardware and the software.

- Level IV: Scalable, Configurable and Multi-Tenant-Efficient
  At this final level, there is a need to scale the application to accommodate more customers. The method to provide scalability is by utilizing a load balancer that will shift the load among all the available web servers.

The question on when to choose a particular maturity model over another, strongly depends on business, architectural and operational needs and on customer considerations. In reference to [9], there are three different models we can considering when making this decision:

- **Business model**: How does the chosen maturity model holds financially? Choosing an isolated environment might be costly for customers, but it might

bring the benefit of privacy that certain type of customers require. This might be a key point to target a group of customers, but it might also be irrelevant to others. Whatever model is chosen it has to be able to bring revenue.

- **Architectural model**: If the application is being built from scratch, it will be easier to make a transformation to a fully mature application. On the other hand, if the application is being migrated from an existing client-server model, there might be some restrictions that would require a large amount of resources to tackle.

- **Operational model**: Customers and SLAs dictate this. Customers can share an environment or customers that are in need of a special disaster-recovery solution

2.4 Case study presentation: Trezone

As a practical application of the KPIs that are introduced in section 4 Trezone will be analyzed.
Trezone is a SaaS application created in the early 2000s that provides an on-line treasury solution accessible through the Internet. It provides a unique environment for corporations to perform operations such as cash flow forecasting, FX trade dealing and guarantee management.

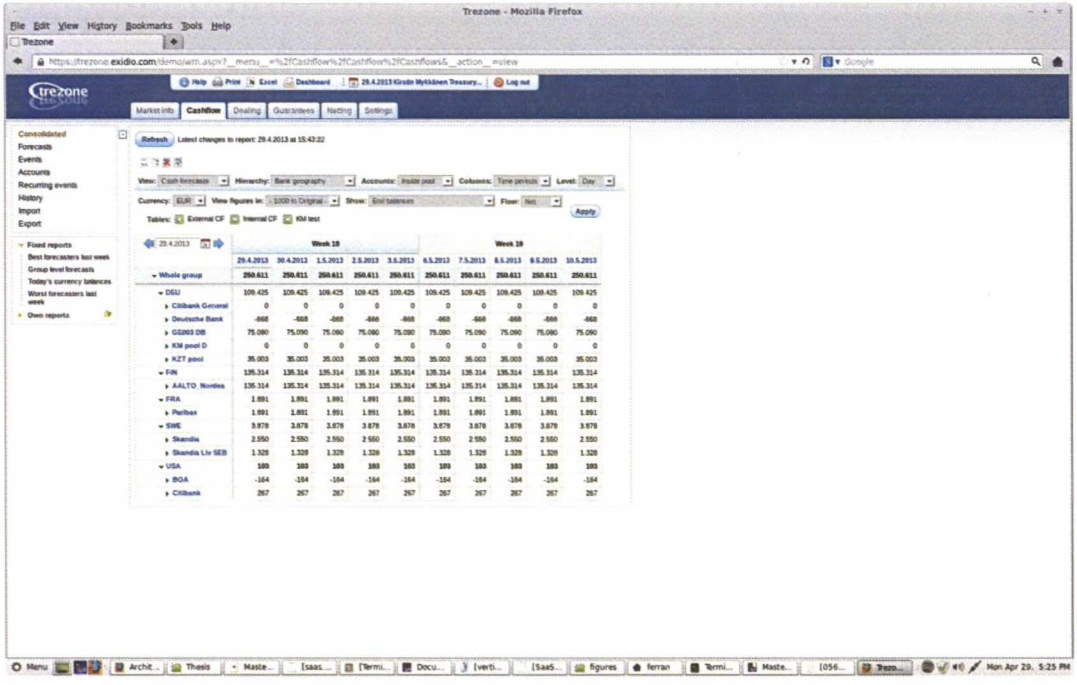

Figure 5: An overview of Trezone.

2.4.1 Technical description

Trezone is a multi-tenant, scalable SaaS application. Due to the fact that the application was built from the scratch and was not ported from shrink-wrapped software, it is optimally built to fully support the maturity model that was previously defined. This becomes clear if we analyze each of the items introduced in the maturity model 2.3.

- Scalability
  Trezone can be vertically and horizontally escalated. This allows to enlarge the system as more customers subscribe to it. As it was pointed out in 2.2.1, if a certain subset of customers taxes more the web servers, the application can be configured to allocate a web server for this set. This is also the case if customers make extensive usage of the database servers.

- Multi-tenancy
  All the customers who utilize Trezone make use of the same application instance. This facilitates the task of adding and maintaining code because there are no differences between the code that different customers execute. Databases share the same schema but they are isolated from each other. The isolation is a necessity for an application that stores sensible data (such as account balances and forecasts).

- Configuration
  Despite all the customers accessing the same application instance and sharing the same code, Trezone allows a certain degree of customization. The user interface can be customized to adapt to customer needs, without having a negative effect on the code maintenance. The customization level is achieved by introducing customizable CSS layouts. Similarly, modules and features can be added or removed to adapt to customer needs, allowing customers to pay for the parts of the application they use.

As it has been described and based on the parameters defined in the maturity model 2.3, Trezone is a good example of SaaS application which makes it suitable to analyze the indicators defined in chapter 4.

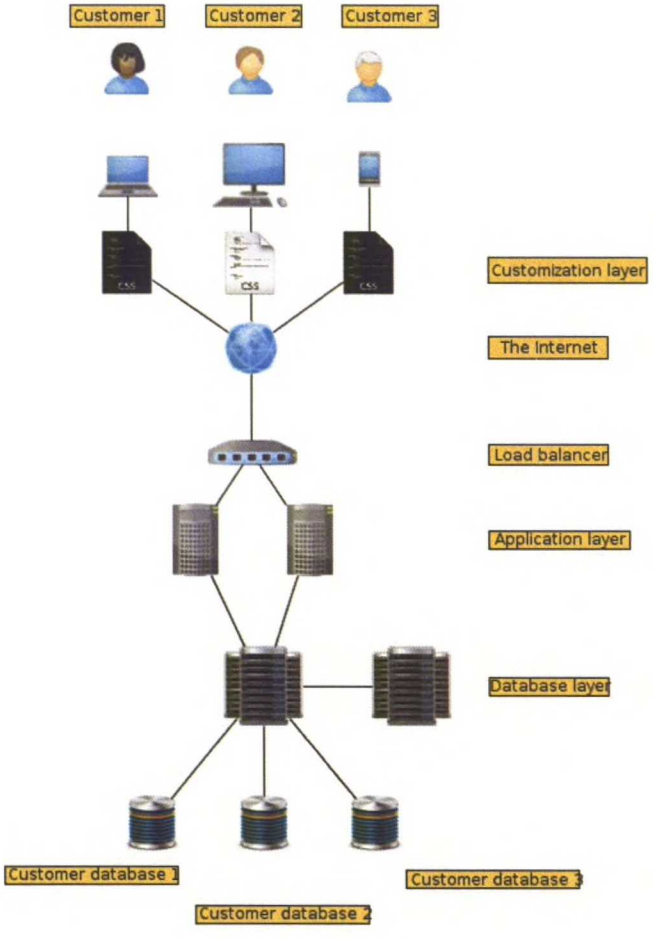

Figure 6: An overview of the architecture of Trezone.

3 Software Quality

3.1 Introduction

In the first part of this chapter, we will introduce the concept of quality in software and the models available to ensure it. In the second part we will introduce the two quality models and the metrics defined in the ISO/IEC 9126. In the final part of this chapter we will present six characteristics that are specific of SaaS applications. These are the characteristics that will be used as a base to define a set of indicators in the next chapter.

3.2 What is Software Quality

Software quality can be defined in multiple ways. According to [16], software quality can be analyzed based on the lack of "bugs" a piece of software has and based on the amount of functional defects it contains. This could expressed in a more formal way:

- Defect rate: The number of defects found per number of lines of code in the software.
- Reliability: The number of failures that occur over time when using the software.

The concept of "fitness for use", is defined by Dr. Joseph Juran and it incorporates the viewpoint of the customer. Essentially it concentrates on fulfilling customer requirements which leads to usage satisfaction, and it mitigates possible feature deficiencies.

To summarize, we can say that quality can be understood by a low number of defects and failures of the software, in addition to meeting the customer needs with it.

3.3 Measuring software quality

In accordance to the Consortium for IT Software Quality (CISQ), five characteristics are desirable in software to provide business value. These are: reliability, efficiency, security, maintainability and size. In order to be able to measure these, we need a model and metrics.

Models provide an overall view of what qualities should be analyzed and metrics provide the tools to quantify it.

According to [8], we can qualify software quality models into three categories based on definition, assessment and prediction. Definition models are used to define what is quality. Assessment models use metric-based approaches to assess the quality of a system. Prediction models are used to predict the possible quality of a system.

For our work, we need a quality model that defines quality and provides metrics to measure it. The ISO/IEC 9126 is a definition model, but it also provides metrics

The Quality in Use Model	
Characteristics	Subcharacteristics
Effectiveness	Effectiveness
Productivity	Productivity
Safety	Safety
Satisfaction	Usefulness

Table 1: The Quality in Use model characteristics and subcharacteristics

to assess the level of quality of software. This is why the ISO/IEC 9126 is a suitable solution for our work and will be chosen to be further discussed.

3.4 The ISO/IEC 9126

The International standard for the evaluation of software quality, ISO/IEC 9126 [12], essentially defines two models composed of several characteristics:

- A quality in use model composed of four characteristics [...] that relate to the outcome of interaction when a product is used in a particular context of use.
- A product quality model composed of six characteristics [...] that relate to static properties of software and dynamic properties of the computer system.

We can therefore say that the quality in use model will be utilized over a system when the goal is to measure interactions. The interactions will occur between users and the system or between administrators and the system. On the other hand, the product quality model will be utilized when there is a need to measure the quality of a computer system that includes software.

3.4.1 The Quality in Use Model

As it has been introduced, the quality in use model focuses on measuring the quality level produced from interacting with a system. This gives an overview of the quality perceived by the customer which is one of the questions software quality should answer. (ISO/IEC 9126-1)

Effectiveness
Effectiveness is defined as the accuracy and completeness with which users achieve specific goals.

Productivity
Productivity relates to the spent resources in relation to the accuracy and completeness with which users achieve goals.

Safety
Safety is the capability of the software product to achieve acceptable levels of risk of harm people, businesses, software property or the environment in a specified context of use.

Satisfaction
Satisfaction is the degree to which user needs are satisfied when a product or system is used in a specific context of use.

3.4.2 The Product Quality Model

The product quality model targets only measuring the quality of a software product. This underlines the overall quality of a software system (ISO/IEC 9126-1).

Functionality
Functionality as seen in figure 7, can be defined as the level or degree to which a product provides functions that meet the needs of a user under certain conditions or in an specific environment. It is divided into three subcharacteristics:

- Suitability: The capability of the software product to provide the right or agreed results or effects with the needed degree of precision. objectives.
- Accuracy: The capability of the software product to provide the right or agreed results or effects with the needed degree of precision.
- Interoperability: The capability of the software product to interact with one or more specified systems.

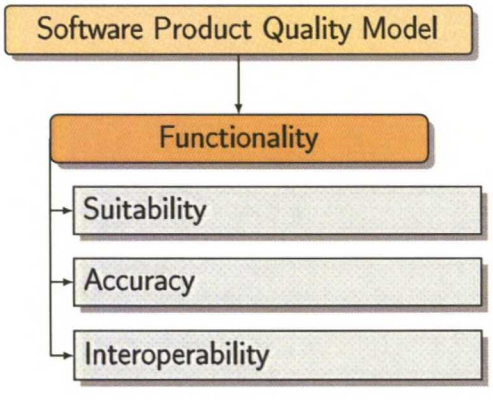

Figure 7: Functional suitability according to the product quality model.

Reliability Reliability as seen in figure 8, is the capability of the software product to maintain a specified level of performance when used under specified conditions. It is defined by three subcharacteristics:

- Maturity: The capability of the software product to avoid failure as a result of faults in the software.
- Fault-tolerance: The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface.
- Recoverability: The capability of the software product to re-establish a specified level of performance and recover the data directly affected in the case of a failure.

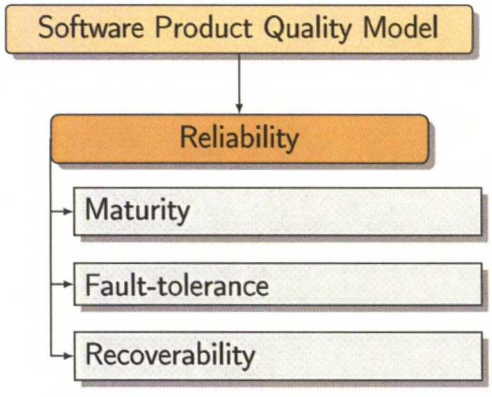

Figure 8: Reliability according to the product quality model.

Usability
Usability as seen in figure 9, is the capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions. It is defined by four subcharacteristics:

- Understandability: The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.
- Learnability: The capability of the software product to enable the user to learn its application.
- Operatibility: The capability of the software product to enable the user to operate and control it.
- Attractiveness: The capability of the software product to be attractive to the user.

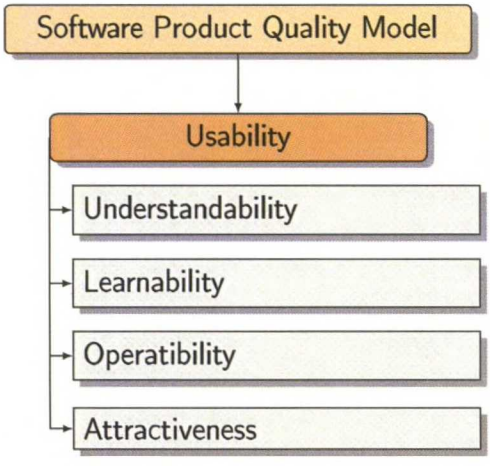

Figure 9: Usability according to the product quality model.

Efficiency
Efficiency as seen in figure 10, is the capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions. Two subcharacteristics define it:

- Time-behavior: The capability of the software product to provide appropriate response and processing times and throughput rates when performing its function, under stated conditions.
- Resource utilization: The capability of the software product to use appropriate amounts and types of resources when the software performs its function under stated conditions.

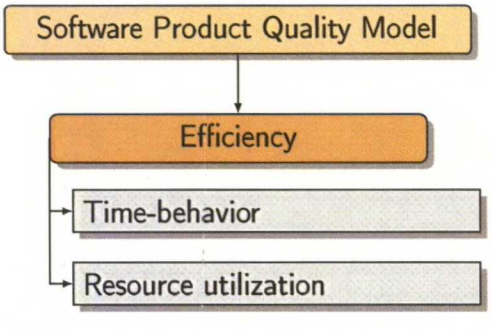

Figure 10: Efficiency according to the product quality model.

Maintainability
Maintainability as defined in 11, is the capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications. Four subcharacteristics define it:

- Analysability: The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.
- Changeability: The capability of the software product to enable a specified modification to be implemented.
- Stability: The capability of the software product to avoid unexpected effects from modifications of the software.
- Testability: The capability of the software product to enable modified software to be validated.

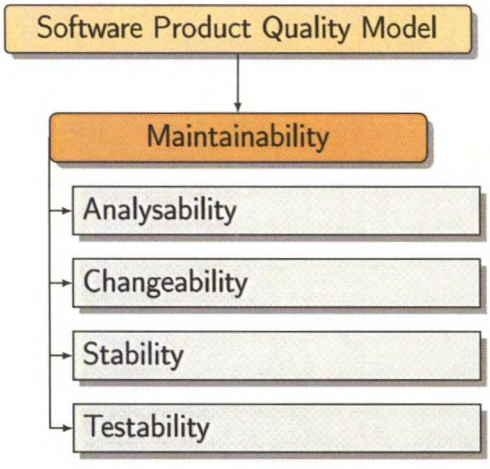

Figure 11: Maintainability according to the product quality model.

Portability
Portability as seen in figure 12, is the capability of the software product to be transferred from one environment to another.

- Adaptability: The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered.
- Installability: The capability of the software product to be installed in a specified environment.
- Co-existence: The capability of the software product to co-exist with other independent software in a common environment sharing common resources.
- Replaceability: The capability of the software product to be used in place of another specified software product for the same purpose in the same environment.

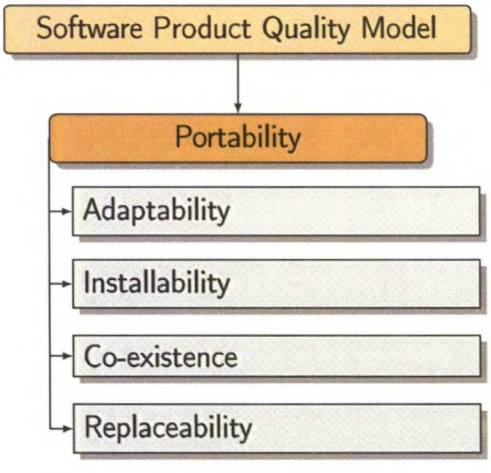

Figure 12: Portability according to the product quality model.

3.5 Metrics in the ISO/IEC 9126

Metrics in the ISO/IEC 9126 are divided into three parts: External and internal metrics of the software and quality in use. Internal and external metrics are categorized into six quality characteristics which subdivide into further subcharacteristics. Internal metrics are applied during the development phase of the software product and are used to predict the quality of the final product.
External metrics, on the other hand, are used to measure the behavior of software when it is being executed in the system environment in which is intended to operate.

Software quality in use contains four characteristics. This model is used to measure whether a product meets the desired needs for a particular set of users. The software quality in use model can only be utilized in a realistic system environment.

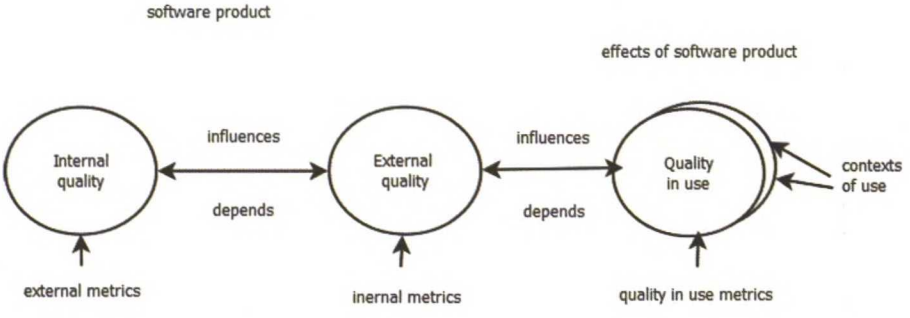

Figure 13: Relationship between metrics and quality

For the scope of this work, we will only concentrate in the external metrics that are needed in section 4. The whole set of internal and external metrics can be found the ISO/IEC 9126-2 [13].

3.5.1 Metric definition

External metrics are categorized into 6 characteristics: Functionality, Reliability, Usability, Efficiency, Maintainability and Portability. (ISO/IEC 9126-2)

Functionality

Functionality measures the functional behavior of a system that contains the software. It includes the following metrics:

- Suitability metrics: Measures the occurrence of an unsatisfying function or the occurrence of an unsatisfying operation during testing and user operation of the system.
- Accuracy metrics: Measures the frequency of users encountering the occurrence of inaccurate matters.
- Interoperability metrics: Measures the number of functions or occurrences of less communicative involving data and commands, which are transferred easily between the software system and other systems or other software products.

- Security metrics: Measures the number of functions with, or occurrences of security problems such as failing to prevent leak of secure information, failing to prevent lost of important data or failing to defend against illegal access or illegal operations.

Reliability

Reliability measures the attributes related to the behavior of the system to indicate the extent of reliability of the software in that system during operation. It includes the following metrics:

- Maturity metrics: Measure attributes such as the software freedom of failures caused by faults existing in the software itself.
- Fault tolerance metrics: Measures the software capability of maintaining a specified performance level in cases of operation faults.
- Recoverability metrics: Measures the attributes as the software system being able to re-establish its adequate level of performance and recover the data directly affected in the case of a failure.

Usability

Usability metrics should asses whether new users can understand if the system is suitable and how it can be used for particular tasks. It includes the following metrics:

- Learnability metrics: Assesses how long users take to learn how to use particular functions, and the effectiveness of help systems and documentation.
- Operability metrics: Assesses whether users can operate and control the software.
- Attractiveness metrics: Assesses the appearance of software and will be influenced by factors such as screen design and color.

Efficiency

Efficiency measures such attributes as the time consumption and resource utilization behavior of a computer system including software. It includes the following metrics:

- Time behavior metrics: Measures the time behavior of a computer system including software.
- Resource utilization metrics: Measures the utilized resources behavior or a computer system including software.

Maintainability

Maintainability metrics measures such attributes as the maintainer, user, or system including the software, when the software is maintained or modified during testing or maintenance. It includes the following metrics:

- Analysability metrics: Measures the maintainer's or user's effort or spent resources when trying to diagnose deficiencies or causes of failures, or for identifying parts to be modified.
- Changeability metrics: Measures the maintainter's or user's effort by measuring the behavior of the maintainer, user or system including the software when trying to implement a specified modification.
- Stability metrics: Measures the attributes related to unexpected behavior of the system including the software when the software is tested or operated after modification.
- Testability metrics: Measures attributes such as the maintainer's or user's effort by measuring the behavior of the maintainer, user or system including software when trying to test the modified or non-modified software.

Portability

Portability metrics should measure such attributes as the behavior of the operator or system during the porting activity. It includes the following metrics:

- Adaptability metrics: Measures such attributes as the behavior of the system or the user who is trying to adapt software to different specific environments. When a user has to apply an adaptation procedure other than previously provided by software for a specific adaptation need, user's effort required for adapting should be measured.
- Installability metrics: Measures such attributes as the behavior of the system or the user who is trying to install the software in a user specific environment.
- Co-existence metrics: Measures such attributes as the behavior of the system or the user who is trying to use the software with other independent software in a common environment sharing common resources.
- Replaceability metrics: Measures such attributes as the behavior of the system or the user who is trying to use the software in place of other specified software in the environment of that software.

The necessary metrics that need to be utilized to measure the attributes that were defined in 3.6 are:

- Reusability and Customizability: In order to analyze the reusability of the application we need to consider three metrics: Analysability, Changeability and Stability.

- Availability: In order to measure the availability of the application, we can utilize the metric proposed to measure Recoverability.
- Data manage by providers: In order to measure this characteristic of SaaS systems, the metrics for reliability will be used.
- Scalability: In order to measure scalability, the metrics for adaptability will be used.

3.6 Measuring Quality in SaaS

The architectural characteristic of SaaS application were defined in 2.2. These were scalability, multi-tenancy and customization:

- Multi-tenancy
  We have seen that one of the architectural feature in SaaS applications is multi-tenancy 2.2.2. This allows for the same application instance to be re-used by multiple customers, creating a one-to-many relationship. The challenge is to determine when, and single instance of an application that is being used by multiple customers, meets the desired satisfaction levels.

- Scalability
  As it was explained in 2.2.1, the vendor must be able to escalate the system if deemed necessary. The ability of a vendor to escalate the application, cannot affect the rest of customers utilizing the application. Customers perceive a lack of scalability by, for example, noticing performance issues.

- Customization
  The configuration level that was previously introduced in 2.2.3 is an example of this feature. If an application is poorly customizable, customers will have to find ways to adapt to it. This might lead to situations where customers find the application too far from their needs, which results in stop using it.

According to [15] there are three more features we can utilize to measure the quality of SaaS applications:

- Availability
  SaaS applications that utilize the Internet as their medium to reach users, require to be available to make use of them. Failing at providing availability traduces in users not being able to utilize it.

- Data Managed by Providers
  The SaaS vendor is responsible for guaranteeing access to the application. This means that long running requests when accessing the application, or poor responsiveness must be avoided. Failing in this category would diminish the quality level that a customer perceives of the application.

- Pay on the go
  This attribute is related to customization, because it allows to adapt to the needs of a customer. A clear example of this comes when a customer is not going to utilize the whole application but just a part of it. A need to adjust the costs for what it is being used is presented, which results in customers perceiving that they pay for what they use.

Having defined the six characteristics we can find of importance in SaaS applications, we can utilize the previously defined models 3.4 and metrics 3.5 to quantify them. In the analysis we will establish relationships between the SaaS characteristics and the metrics necessary to quantify them, the output will be the set of indicators that we strive to provide in this thesis.

4 Key Performance Indicators

4.1 Introduction

This chapter provides a set of key performance indicators for software as a service applications. They are based on the models 3.4 and the metrics 3.5 in conjunction with the characteristics defined for SaaS applications in 3.6.

4.2 A KPI for Efficiency

Efficiency is defined as the amount of spent resources in relation to the accuracy and completeness with which users achieve specific goals 3.5.1. This means that when a user interacts with a system expecting to achieve a result, the user does not face long running requests nor errors.

From the technical point of view, efficiency is seen as the application consuming little resources to provide service. For instance, if the algorithms utilized in parts of the application are efficient, the application will consume little resources to execute them. It also relates to how responsive the application is and how long it takes for a user to access it.

From these two aspects, we can deduct that we need to measure resource utilization and response times. From the previously defined SaaS-related attributes 3.6, the "data managed by providers" and scalability belongs in this KPI.

Scalability problems will manifest when users notice a lack of performance in the application which is related to the time behavior of the system and the amount of available resources.

4.2.1 Time behavior

The time behavior measures the response time of a computer system.

For SaaS applications, these times represent the amount of time it takes for a server to respond to a request. A high response time indicates that the application takes high amounts of time to process requests, causing users to wait and get a feel of sluggishness. There are several factors that can affect response times such as resource utilization being high or network connectivity issues.

On an article written by Jakob Nielsen from the Nielsen Normal Group [17], it is pointed out that a response time of up to one second, is considered good for utilizing a web site. Anything above that value, it is considered to keep the user at the mercy of the computer.

We can therefore say that, to achieve a good level of response time, they should never go above the 1 second threshold.

4.2.2 Resource utilization

Resource utilization measures the utilized resources behavior of the system. Memory utilization, CPU and I/O devices utilization fall into this category and they all can be used to calculate the overall resource utilization of a system. As an example, we

will present the proposed formula extracted from the metrics in 3.5.1, that relates to the amount of memory-related errors that occur over a period of time.

$$\text{Memory utilization} = \frac{A}{B} \tag{1}$$

where A is the number of memory fault messages and B is the user operating time during user observation. The desired output value is as low as possible because it indicates there are less errors derived from memory failures.

For I/O devices utilization we can apply a similar metric:

$$\text{I/O utilization} = \frac{A}{B} \tag{2}$$

where A is the time amount of warning messages or failures and B is the operating time during observation. The desired output value is as low as possible because it indicates there are less errors derived from I/O devices failures.

If the memory utilization measurement shows a high amount of memory fault messages, this means that the system is lacking memory and it should be increased. If the I/O devices show a high amount of failures, it indicates that the devices are too busy. For CPU utilization, the value should stay below 80%. As it is explained in [2], a consistent state of 80-90% of processor utilization indicates a need to update the CPU or to add more processors.

4.3 A KPI for Reliability

Reliability is defined as the attributes related to the behavior of the system when it is in use 3.5.1. We can analyze two aspects: stability and correctness of a system.

As we have seen for SaaS applications, multi-tenancy 2.2.2 explains that users share the same application instance. This means that when faults or failures occur in that software, they might propagate to other customers that share the same instance. In a stable system, the amount of faults and failures should be counted as low as possible. System correctness means that the user will not get error messages when accessing the application. For SaaS applications this can be seen as the codes returned by the sever when users requests resources from the server.

4.3.1 System stability

System stability includes the amount of faults and failures that occur in a system. Failures are caused by faults in software and they manifest by not allowing a user to use the affected part to perform an action that would otherwise be available. Faults, on the other hand, are invisible to the user but they might lead to potential failures.

In order to measure the amount of faults, we can use the following metric proposed in 3.5.1:

$$X = \frac{D_f}{T} \tag{3}$$

where D_f is the amount of detected faults and T is the total amount of faults and failures.

Similarly we can measure the amount of failures:

$$Y = \frac{D_F}{T} \tag{4}$$

where D_F is the amount of detected failures and T is the total amount of faults and failures.

In both cases, for a stable system the result should be as close to zero as possible.

4.3.2 System correctness

System correctness responds to how correct the data produced by the system is from the point of view of the user. When a user reaches the system and performs an action, how many of the responses the user received are correct. In order to measure the amount of faults, we can use the following metric proposed in 3.5.1:

$$X = \frac{A}{B} \tag{5}$$

where A is the amount of correct responses received by the user and B is the total amount of requests performed by the user. The higher the value is close to one, the more correct the system is. For a SaaS application, we can utilize the status codes returned by the server to measure it. As it is described in [5], 2XX status codes mean a successful requests and 4XX refer to errors. We can substitute the A value in 5 with the amount of 2XX codes to obtain the amount of successful requests.

Availability	Downtime per year	Downtime per month	Downtime per week
0.90	36.5 days	72 hours	16.8 hours
0.95	18.25 days	36 hours	8.4 hours
0.97	10.96 days	21.6 hours	5.04 hours
0.98	7.30 days	14.4 hours	3.36 hours
0.99	3.65 days	7.20 hours	1.68 hours
0.995	1.83 days	3.60 hours	50.4 hours

Table 2: Comparison of availability and downtimes

4.4 A KPI for Availability

Availability is defined as the total time the application is available for use. This includes the times when the application was unavailable due to maintenance or due to system recovery processes. From the previously defined SaaS-related attributes 3.6, the availability item belongs in this KPI.

The proposed formula to measure the availability of a SaaS system, is taken from the metrics defined for recoverability in 3.5.1:

$$\text{Availability} = \frac{T_o}{(T_o + T_r)} \tag{6}$$

where T_o is the total operation time and T_r is the time the system was unavailable. The closer to one is the result, the higher available the system is.

Highly available systems should strive for a result as close as one as possible. Table 2 table translates the availability in into year, month and weeks downtimes.

4.5 A KPI for Adaptability

Adaptability responds to how well the application adapts to two of the characteristics that define SaaS systems 3.6: Reusability and Customizability.

Reusability responds to use the same application instance by multiple customers and customizability responds to adapting the application to user's needs by customizing it.

In order to measure the level of reusability, we need to measure the parts of the application that need to be modified in order to be usable by other customers as well as the amount of generated malfunctions as a result of implementing changes to customize the application. For changeability, we need to measure the complexity of implementing these changes.

4.5.1 Reusability

We can measure reusability by using the following metrics:

$$\text{Reusability} = \frac{P_c}{T_p} \tag{7}$$

where P_c are the parts of the application that need to be changed and T_p are the total parts of the application. The desired value resulting from applying this equation would be as close to zero as possible.

In order to conclude measure reusability, we consider the amount of generated malfunctions after introducing the changes to the application.

$$\text{Generated errors} = \frac{N_a}{T_a} \tag{8}$$

where N_a is the number of turns which user encounters failures during operation after the software was changed. T_a is the operation time during the specified observation period after the software was changed.

4.5.2 Changeability

$$\text{Changeability} = \frac{\sum \frac{A}{B}}{N} \tag{9}$$

where A is the work time spent on implementing the change, B is the amount of files changed and N is the number of changes.

The desirable output is as short as possible, because it reflects little effort to modify the software.

5 Results

5.1 Tools to acquire data to measure KPI

In order to acquire the necessary data to be used in the indicators defined in the previous section, we will introduce the following tools:

Microsoft IIS server logs

Software as a service applications that rely on Microsoft IIS, are able to configure the web server to enable W3C logging fields. The following fields would need to be enabled to measure time behavior:

- Method (cs-method): This field provides information on the method utilized. The W3C defines the following methods [5]: GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT.
- URI Stem (sc-uri-stem): It provides the Universal Resource Identifier of the action performed by the client.
- URI Query(sc-uri-query): It provides the query that the client was trying to perform. This is only valid for dynamic pages.
- Protocol Status (sc-status): The protocol status HTTP or FTP code.
- Protocol Sub-Status (sc-substatus): The protocol substatus for HTTP or FTP protocol.
- Time taken (time-taken): The length of the action expressed in milliseconds.

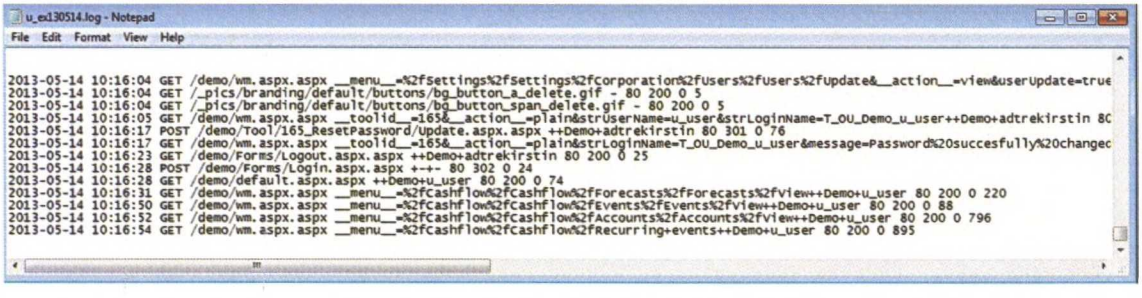

Figure 14: An extract of the information Microsoft IIS server logs provide

The data generated from this tool is necessary in the following KPIs:

- Efficiency, needed for the time behavior.
- Reliability, needed for the system correctness.

Windows Performance Monitor

Giving a deeper explanation of this tool falls out of the scope of this work, therefore we will quote Microsoft regarding the definition of the performance monitor. Extracted from the Microsoft website:

> "Performance Monitor provides a visual display of built-in Windows performance counters, either in real time or as a way to review historical data. You can add performance counters to Performance Monitor by dragging and dropping, or by creating custom Data Collector Sets. It features multiple graph views that enable you to visually review performance log data. You can create custom views in Performance Monitor that can be exported as Data Collector Sets for use with performance and logging features."

Extract of the information it provides:

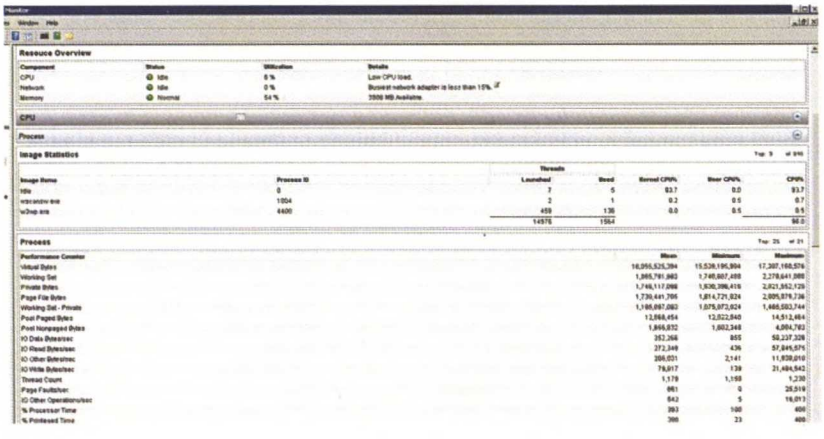

Figure 15: A snapshot of the information the performance monitor provides.

The data generated from this tool is used in the resource utilization in the efficiency KPI.

Bug tracking system

Explaining what an issue tracker falls out of the scope of this work, therefore a formal definition from Wikipedia will be utilized:

> "A bug tracking system or defect tracking system is a software application that is designed to help keep track of reported software bugs in software development efforts. It may be regarded as a type of issue tracking system."

An example of this tool is Bugtracker.NET [1]. It allows customer support to log items such as: customer requests to modify the system, system errors reported by system users and system unavailability reported by system users.

When users encounter a defect in the application or are in need of a modification to adjust the application to their needs, they contact customer support who in turns

fills out the necessary information. This creates cases which contain the necessary information to measure the adaptability and reliability KPIs.

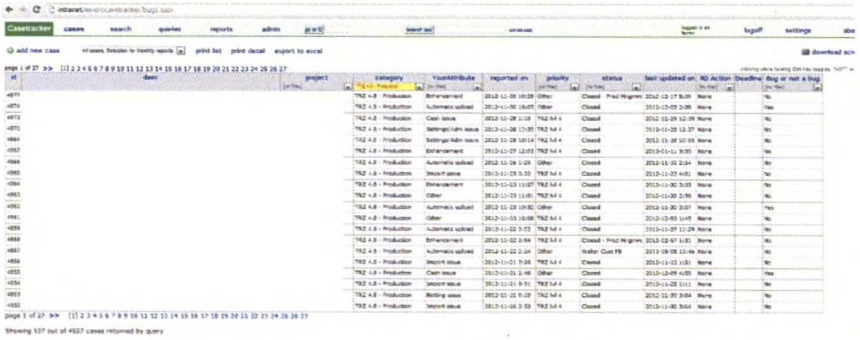

Figure 16: Example of the data the issue tracker provides.

From this tool we can extract the necessary data to be used in:

- Adaptability, needed in reusability and customizability
- Availability

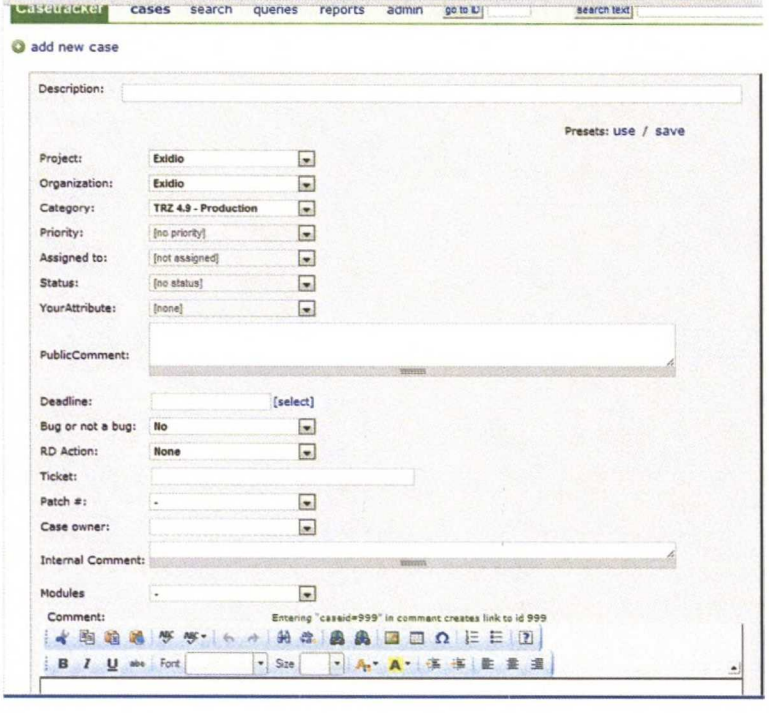

Figure 17: Example of the necessary information when opening a new case.

Splunk

Splunk [3] is a tool that provides monitoring and reporting tools to analyze logs. It supports Microsoft IIS Server log parsing as well as Microsoft windows events log parsing.

For the purpose of this work, it has been utilized when parsing IIS logs. This provided the information for the time behavior in the efficency KPI, and in the system correctes in the reliability KPI. In picture 18, it is shown how splunk can provide the necessary information to measure the time behavior of a system.

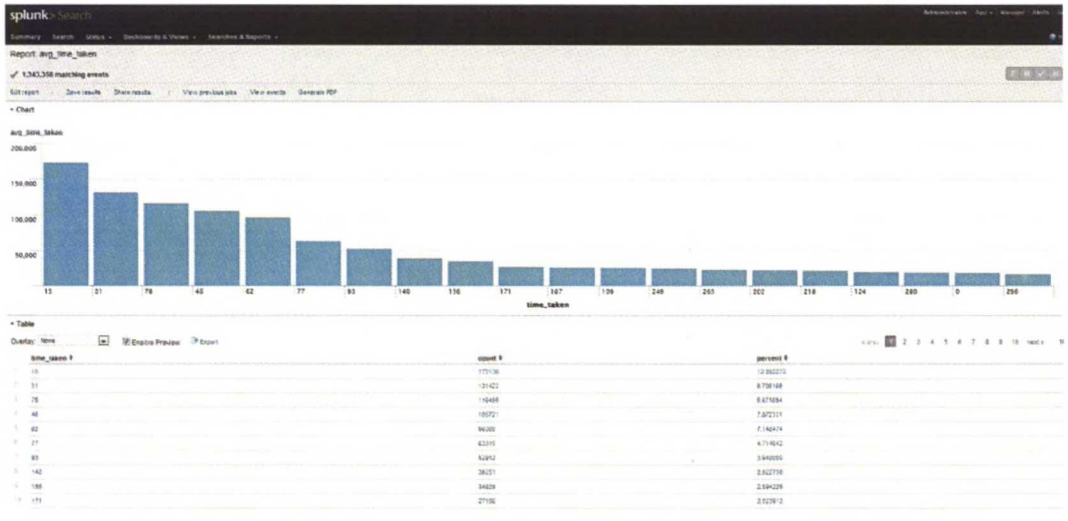

Figure 18: A snapshot of the information Splunk provides.

Time taken	Detailed	Percent
<1s		95,8
	0-250 ms	77,1
	251-500 ms	16,3
	501-1000 ms	2,4
<2s		2,6
<3s		0,8
>3s		0,8

Table 3: Average time taken to process requests in Trezone.

5.2 Efficiency in Trezone

The items we should analyze in Trezone to determine the efficiency, have been defined in 4.2. In this section we will analyze them with the information extracted from Trezone.

Time behavior

The analyzed data corresponds to a month of Trezone usage in one of the web servers. A total amount of 1,343,358 entries were parsed using Splunk 5.1. The results are displayed in table 3. As it is seen from the results in Trezone, the majority of response times concentrate below 500 ms which is considered a good result according to the information from Jakob Nielsen in [17].

Resource utilization

The result presented regarding the amount of resources utilized during that time period, corresponds to the processor utilization of all the servers that host Trezone. Unfortunately there is no data available to calculate the I/O devices nor memory utilization, because it was not enabled in any of the servers. If the monitoring of I/O devices and memory is enabled, this information can be extracted in a similar way as it is done for the processor usage.

The data related to the processor usage is captured during the same month that the time-behavior metric was analyzed. Figures 19, 20 show the CPU utilization for the web servers and figures 21, 22 show the CPU utilization for the database servers.

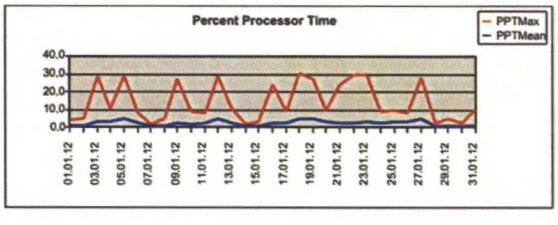

Figure 19: Web server 1 CPU resource utilization.

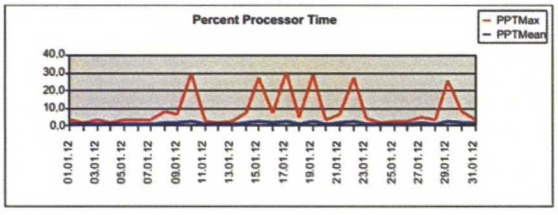

Figure 20: Web server 2 CPU resource utilization.

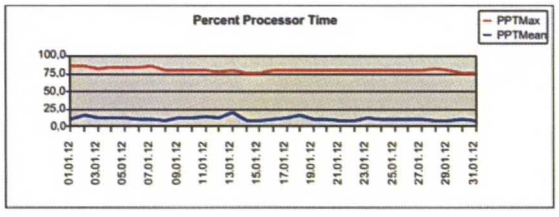

Figure 21: Database server 1 CPU resource utilization.

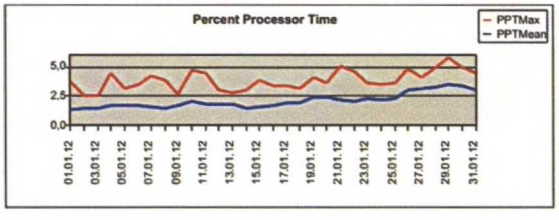

Figure 22: Database server 2 CPU resource utilization.

As it is seen from the graphs provided by the performance monitor, the amount of processor utilized in the web servers peaks at 30%. This is considered as a low value and does not show a need for upgrading the servers. For one of the database servers, however, there is a constant utilization of CPU as high as 75%. This value can lead to problematic situations if, at give times, the CPU utilization peaks at more than 90%. As it is explained in [2], a consistent state of 80-90% of processor utilization indicates a need to update the CPU or to add more processors.

5.3 Reliability in Trezone

In order to gather the necessary data to measure stability, we will utilize the data collected in the logs of the application.
In the case study of this work, the IIS logs will provide the necessary information. This is because they provide the information related to user interaction with the system.
The application logs will include exceptions when a user performs an action that leads to a fault, and application errors will be displayed when that fault leads to a system failure.
The server logs will provide the necessary information regarding the system availability. This logs includes information regarding the requested pages and the server response.
The items we should analyze in Trezone to determine the reliability, have been defined in 4.3. In this section we will analyze them with the information extracted from Trezone.

System stability

Unfortunately real data could not be obtained to analyze this part in Trezone. Therefore we will show how this could be calculated if we would have the necessary data. In order to count the amount of faults that appear in Trezone, we will consider faults exceptions that are catched by the application. Exceptions that are catched will not crash the application and will be invisible to the end user. In figure 23 there is an example of what could be considered a fault in Trezone. This fault or exception is catched and does not crash the application.

In order to count the amount of failures that appear in Trezone, we will consider failures as exceptions that are not catched, or any other errors that crashes the application. Failures are visible to end users. Figure 24 shows how an application failure is displayed in the logs. This type of error messages are visible to the user and crash the application momentarily.

In order to obtain the final calculation, we would need to measure the amount of faults and failures over a period of time. Once this data is available, we can apply the two proposed equations for faults (3) and failures (4).

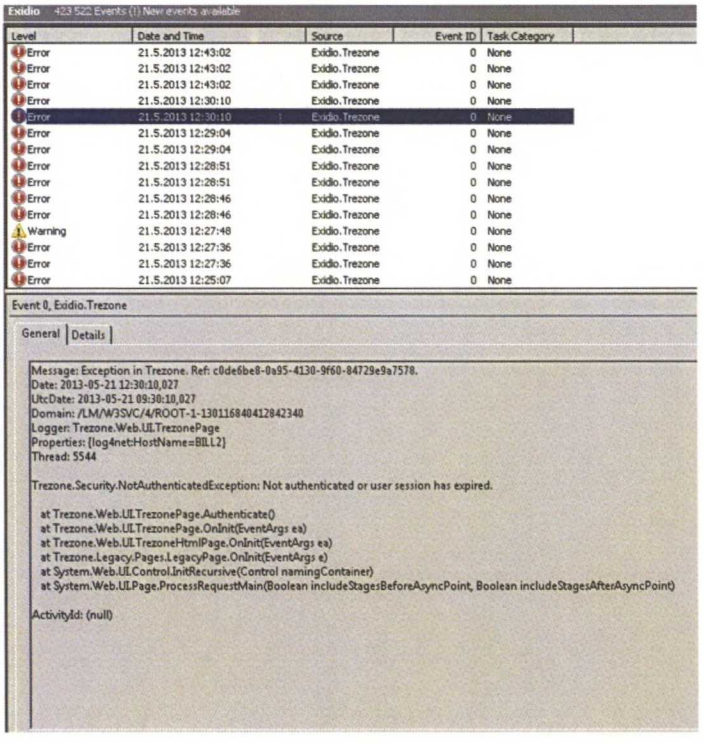

Figure 23: Example of catched exception in Trezone.

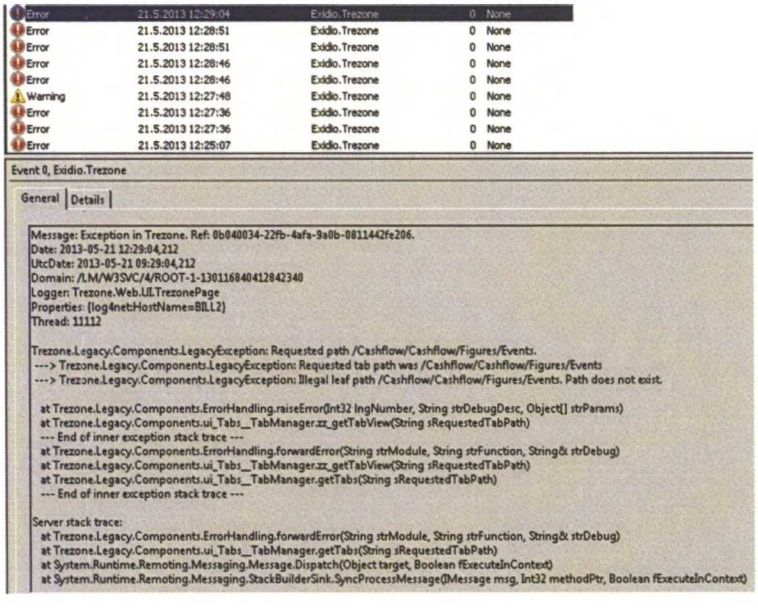

Figure 24: Example of application failure in Trezone.

Status code	Percent
304	54,7
200	42,5
404	0,2
Other	2,6

Table 4: Status codes returned in Trezone

System correctness

The analyzed data corresponds to a month of Trezone usage in one of the web servers. Table 4 shows the results of analyzing a total amount of 1,343,358 entries using Splunk. As it is seen, the majority of returned codes from Trezone are 304 and 200. The code 304 indicates that the resource has not changed and there is no need to download a newer version. The code 200 indicates that the requested resource is available and will be served to the requester. The code 404 indicates that there were errors when serving the request.

Since the majority of displayed codes are 304 and 200, we can conclude the level of correctness is satisfactory. This indicates that the majority of user requests are being served by the server.

5.4 Availability in Trezone

The items we should analyze in Trezone to determine the availability, have been defined in 4.4. In this section we will analyze them with the information extracted from Trezone.
The availability data corresponds to the time span of one release of the product. It has been obtained from the issue tracker that is used to track issues in Trezone 25.

For Trezone version 4.9, between the time period of May 31th, 2012 and 21th of May 2013, a total amount of 5 issues were related to system unavailability. By analyzing each individual reported case, the total unavailability time of the system was 7 hours and 30 minutes. The total amount of operative hours is 8505 (from 31/05/2021 until 21/05/2013).

With these values, we obtain an availability of 0.999.

$$\text{Availability} = \frac{8505}{(8505 + 7.5)} \tag{10}$$

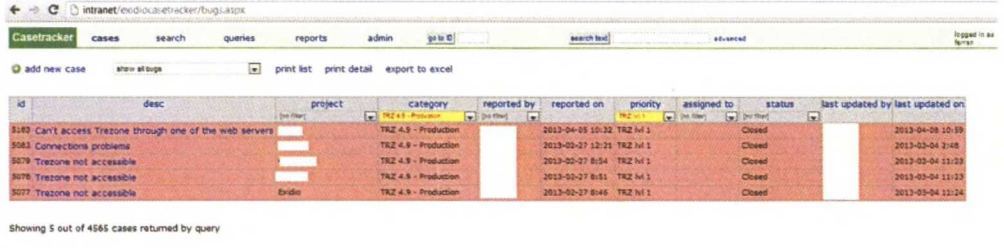

Figure 25: Cases showing availability problems in Trezone.

5.5 Adaptability in Trezone

The items we should analyze in Trezone to determine the adaptability, have been defined in 4.5. In this section we will analyze them with the information extracted from Trezone.

Reusability

The necessary data to calculate reusabilty has been extracted from the issue tracker that was previously defined in 5.1. It takes into account all the enhancements that were necessary to satisfy customers needs during the span of one release of the product (approximately 6 months).

A total amount of 537 issues were reported by users of the system. These issues include system malfunctions (such as bugs), problems related to malfunctions in the environment and enhancements. Out of the 537 issues reported, 75 are enhancements requested by users. General issues are issues such as system malfunctions, application bugs, system unavailability and things of that category. Enhancements are requested to accommodate user needs with the system. In result, around 12% of the reported issues are enhancements.

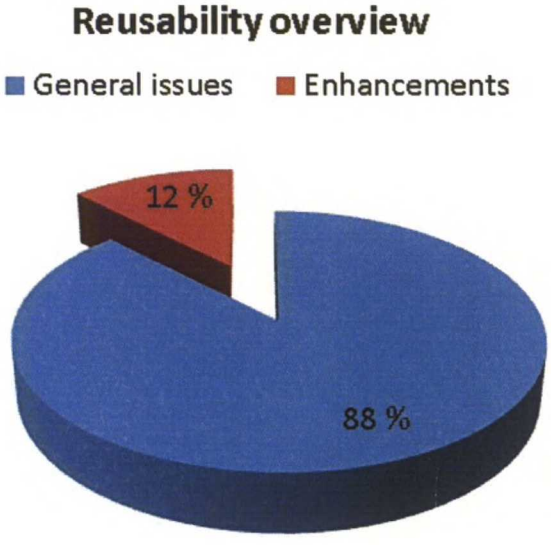

Figure 26: Reusability in Trezone.

The second part of measuring reusability is to find the amount of generated defects after the enhancements have been implemented. The necessary data to calculate this has been extracted from the issue tracker that was previously defined 5.1. It represents all the system malfunctions derived from implementing the requested enhancements. A total of 40 system malfunctions were created as a consequence of enhancing the system. The 75 enhancements implemented resulted in 40 new system malfunctions, which translates into one new bug for every two enhancements.

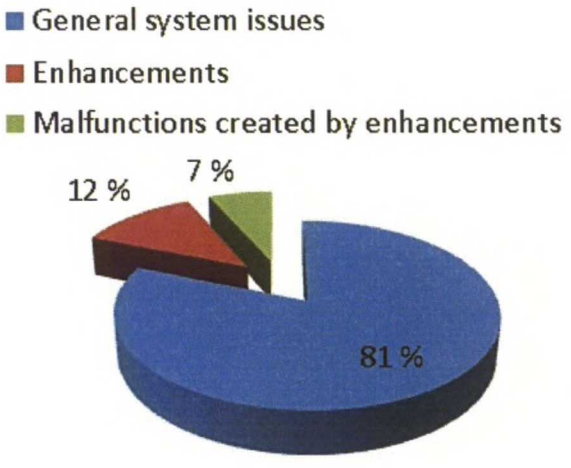

Figure 27: System malfunctions derived from enhancements in Trezone.

Changeability

Unfortunately I could not obtain enough precise data to measure this value. This is because the information relative to the amount of time that a developer needed to fix a system malfunction, was not present anywhere. The only available data was the time a malfunction was reported and when it was fixed.

In order to collect the necessary data to measure changeability, we should establish a process in which developers make an annotation of the amount of time spent on a system modification.

6 Summary

6.1 Conclusions

As it was shown in section 5, the proposed key performance indicators proposed in this work can be utilized to quantify the performance of a SaaS application based on the typical aspects of these applications in conjunction with ISO metrics. This was the initial goal of this thesis and it has been reached.

There are however, three aspects that would need to be implemented in order to apply these indicators more efficiently.

Firstly, define a process to automatically collect the necessary data to be used with the KPIs formulas would be needed. This process would have to be adapted to each SaaS application, because although SaaS applications share similar characteristics, the technologies utilized to implement them are different. For instance, we can find SaaS applications hosted on the Microsoft IIS Server but also on Apache servers. The generated data for these two servers will be similar, but different enough so that it will have to be adapted.

Secondly, a definition of what is an acceptable and unacceptable value for each KPI. We have seen that for instance, the time behavior in Trezone is 95% of time below 1 second. For Trezone, we can conclude that this is a good value because it is fast enough for users to make use of it without getting a feeling of being at the mercy of the computer. However, there might be SaaS applications which might require even faster response times. In this situations, the KPIs could be used to determine what aspects of the system are not performing as they should.

Finally, to create a software implementation of these indicators. Something similar to what Splunk [3] has achieved but on a lower scale. This tool could utilize the two process that were mentioned earlier (automatically collect data, and define acceptable and unacceptable values) in order to facilitate the task of measuring the performance.

To summarize, the proposed KPIs fit the purposes of measuring the performance of different aspects of a SaaS application. However three aspects could be utilized to optimize their measurement such as: define an automated process to collect data, define what acceptable values for this KPIs and create a software implementation of the KPIs.

References

[1] Bugtracker .net. http://ifdefined.com/bugtrackernet.html. [Online; Accessed 2013.05.15].

[2] Monitor cpu usage. http://msdn.microsoft.com/en-us/library/ms178072.aspx. [Online; Accessed 2013.05.16].

[3] Splunk. http://www.splunk.com/view/splunk/SP-CAAAG57. [Online; Accessed 2013.05.21].

[4] The incredible growth of the Internet since 2000. http://royal.pingdom.com/2010/10/22/incredible-growth-of-the-internet-since-2000/. [Online; Accessed 2013.05.17].

[5] W3C status codes. [Online; Accessed 2013.05.16].

[6] Michael A. Cusumano. Cloud computing and saas as new computing platforms. *Communications of the ACM*, 53(4):3, 2010.

[7] Exidio. http://www.exidio.com/pages/en/Exidio/index.html. [Online; Accessed 2013.04.30].

[8] Klaus Lochmann Stefan Wagner Florian Deissenboeck, Elmar Juergens. Software quality models: Purposes, usage scenarios and requirements.

[9] Gianpaolo Carraro Frederick Chong. Architecture strategies for catching the long tail. http://msdn.microsoft.com/en-us/library/aa479069.aspx#docume_topic2, April 2006. [Online; Accessed 2013.04.30].

[10] Roger Wolter Frederick Chong, Gianpaolo Carraro. Multi-tenand data architecture. http://msdn.microsoft.com/en-us/library/aa479086.aspx, June 2006. [Online; Accessed 2013.04.30].

[11] Kevin Holmes. Grow or add? saas application scaling basics. http://www.rackspace.com/blog/grow-or-add-saas-application-scaling-basics/, August 2012. [Online; Accessed 2013.04.30].

[12] IEEE. ISO/IEC 9126-1.

[13] IEEE. ISO/IEC 9126-3.

[14] Panagiotis Sfetsos Ioannis G. Stamelos. *Agile Software Development Quality Assurance*. Premier Reference Source, 2007.

[15] Du Wan Cheun Soo Dong Kim Jae Yoo Lee, Jung Woo Lee. A quality model for evaluating software-as-a-service in cloud computing. *IEEE Computer Society*, 1:6, 2009.

[16] Stephen Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Professional, 2002.

[17] Jakob Nielsen. Website response times. `http://www.nngroup.com/articles/website-response-times/`.

[18] SIIA. Software-as-a-service; a comprehensive look at the total cost of ownership of software applications. *Software & Information Industry Association*, 1:31, 2006.